Lightweight Kernel Support for Direct Shared Memory Access on a Multi-Core Processor

Ron Brightwell
Sandia National Laboratories*
PO Box 5800
Albuquerque, New Mexico
rbbrigh@sandia.gov

ABSTRACT

This paper describes an enhancement to the Catamount lightweight kernel for direct shared memory access between processes running on a multi-core processor as part of a parallel application. Unlike traditional shared memory support for interprocess communication, which involves dynamic memory allocation and mapping, we leverage Catamount's static contiguous memory mapping scheme to allows the processes on a multi-core process to directly access each other's memory through simple virtual address manipulation. This paper describes our implementation of direct shared memory access in Catamount, contrasts it to other approaches, and discusses the potential benefits of this approach in supporting MPI applications.

1. INTRODUCTION

The Catamount [6] lightweight kernel is a third-generation compute node operating system that was initially developed by Sandia National Laboratories and Cray, Inc., for the Sandia/Cray Red Storm [4] massively parallel processing machine. Red Storm is the prototype for what has become the commercially successful Cray XT line of large-scale, distributed memory parallel computing platforms. The current Red Storm machine contains more than twelve thousand dual-core AMD Opteron processors for a total of nearly twenty-six thousand cores. A significant portion of the machine will soon be upgraded to quad-core processors, bringing the peak performance up to 250 teraflops.

Catamount differs in many ways from traditional UNIX-based operating systems. In particular, all Sandia lightweight kernels were designed to support a limited set of hardware running a relatively small set of applications using a message-passing based programming model. These differences have allowed us to make design decisions and implementation choices that would not be appropriate for general-purpose operating systems. Since we are focused on optimizing performance and scalability for MPI-based modeling and simulation applications for which a significant resource investment has been made, there is little opportunity to explore alternative strategies that would involve significant changes to the programming model or system architecture.

However, the onset of multi-core processors is impacting many of the fundamental design choices that both application programmers and system software developers have made. Applications that have been running and scaling well on tens of thousands of processors using an "MPI everywhere" model are now considering moving to mixed-mode programming models where MPI is used for task-level parallelism between nodes, but Open MP compiler directives or POSIX threads are used for finer-grain parallelism within nodes. The main motivation for this consideration is not the mulit-core processors themselves, but the inability of memory subsystem performance and network performance to keep pace with the increased computational capability. However, experiences with mixed-mode programming approaches on symmetric multi-processors yielded little success. Application developers added significant complexity to their codes to avail of multi-level parallelism and few saw any significant performance improvements.

MPI exacerbates the memory subsystem problem. The MPI model mandates copying data between separate address spaces. While this approach has some advantages in terms of enforcing locality, memory-to-memory copies between processes on a multi-core processor puts even more pressure on the memory subsystem. More importantly, the most efficient way to implement intra-node MPI communications is to use POSIX shared memory, where a region of shared memory is allocated and mapped into each process' address space. For MPI, this means that the sender must copy data into the shared region and the receiver must copy it out. Instead of a single copy, there are two. Alternative approaches have been proposed that allow the operating system or an intelligent network interface to perform a single copy, but all of the approaches have significant drawbacks. See [3] for a comprehensive analysis of the tradeoffs for implementing MPI intra-node communication operations.

In this paper, we describe an approach that leverages Catamount's existing memory management design to implement fixed offset virtual addressing to allow for direct shared memory access between the processes running on a multi-core processor.

*Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

2. BACKGROUND

The Cougar lightweight kernel [9] from which Catamount was derived had support for dual-processor compute nodes [7]. When the parallel application was launched, the user could choose one of three different modes for using the two processors on each node. In message co-processor mode the system processor was dedicated to servicing incoming network messages and handling requests from the user process running on the second processor. This mode improved network latency performance by avoiding the need for interrupts to service the network DMA engine. In application co-processor mode, the system processor ran the kernel as well as the main application process. The application could run co-routines, a limited threading environment, on the second processor. Most applications did not use this capability directly, since it was available through a non-portable interface. Rather, it was enabled in the underlying computational libraries and through Open MP compiler directives. The final mode, virtual node mode, simply treated the two processors as two separate nodes and ran a process on each processor. Cougar had no support for shared memory, so the two processes sharing a node communicated via message passing.

Virtual node mode was the most popular method of utilizing dual-processor nodes because it did not require any changes to existing MPI applications and delivered reasonable application performance. No attempts were made to extend Cougar to support more processors, since the ASCI Red [10] platform on which it ran only supported two processors per node. When Sandia and Cray began the development of Catamount for Red Storm, it was not clear that there would be a need for supporting multiple processors per node. The original design for Red Storm called for four single-processor nodes per board, and Cray had no plans to connect these processors together to create a four-processor nodes. Despite this, Sandia wanted to keep the processor mode functionality from Cougar in Catamount because the overhead of the extra code was minimal and removing the code induced extra risk into a what had been a very stable, production-hardened kernel. Nevertheless, Sandia removed support for multi-processor nodes at the behest of Cray.

This lack of foresight become a significant limitation for Catamount when AMD announced dual-core processors shortly after the work to remove multi-processor support was completed. Sandia then began work to re-integrate virtual node mode support back into Catamount for dual-core Red Storm nodes. It was decided not to implement the other processor modes since Red Storm had an intelligent network interface, call the SeaStar [2], that eliminated the need for message co-processor mode and few applications ever used application co-processor mode. As with Cougar, Catamount only supported interprocess communication through message passing.

Rather than take full advantage of the programmability of the SeaStar, Cray chose to implement message passing using an interrupt-driven implementation of the Portals [1] data movement layer. When a message arrives at the SeaStar, it interrupts the host processor and lets the operating system determine the ultimate destination in user space. The OS then programs the SeaStar DMA engines appropriately. For intra-node transfers less than 512 KB, the OS uses a memory-to-memory copy to move messages between the processes. For larger messages, the OS uses the network DMA engines. This approach proved to be sufficient for dual-core processors.

However, several significant problems arise when using this approach for quad-core processors. Serialization of all intra-node messages through the OS creates a bottleneck. This problem is compounded by the interrupt-driven network stack, resulting in a significant amount of work for one of the cores, and creating a potential load imbalance across the cores. Having multiple paths for intra-node messages – one through shared memory and the other through network DMA – induces added complexity to ensure proper ordering of messages. These problems only get worse as core count increases.

With the impending Red Storm upgrade to quad-core processors, we decided to reevaluate the ability of Catamount to support using shared memory for intra-node data movement in a way that would allow for efficient MPI transfers.

3. DIRECT ACCESS SHARED MEMORY

Catamount does not support demand-paged virtual memory. It uses a linear mapping from virtual addresses to physical pages of memory. This approach can potentially have several advantages. For instance, there is no need to register memory or "lock" memory pages involved in network transfers to prevent the operating system from unmapping or re-mapping pages. The mapping is done at process creation time and is never changed. This can greatly simplify translation and validation of virtual addresses for the network interface. Virtual address validation is a simple bounds check and translating virtual addresses to physical addresses is a simple offset calculation. Sandia leveraged this capability to implement a completely offloaded implementation of Portals for the SeaStar [2].

Our approach for intra-node direct shared memory access takes advantage of this simplified memory management model, specifically the fact that Catamount only uses a single entry in the top-level page table mapping structure (PML4) on each Opteron core. Each PML4 slot covers 39 bits of address space, or 512 GB of memory. Normally, Catamount only uses the first entry, covering physical addresses in the range `0x0` to `0x007FFFFFFFFF`. The Opteron supports a 48-bit address space, so there are 512 entries in the PML4.

To implement direct access shared memory, each core writes the pointer to its PML4 table into an array at core 0 when a new parallel job is started. Each time the kernel enters the routine to run a context, it copies all of the PML4 entries from each core into the other cores. This allows every core on a node to see every other core's view of the virtual memory space across the node, at an easily computed offset into its own virtual address space. The kernel-level code for this is shown in Figure 1(a). Figure 1(b) shows the user-level function that converts a "local" virtual address into a "remote" virtual address on a different core.

Another aspect of Catamount memory management is that the mapping of virtual addresses for the same executable image is identical across all of the processes on all of the nodes. The starting address of the data, stack, and heap is the same. This means that the virtual address of variables with global scope is identical everywhere. The Cray SHMEM get/put environment first available on the Cray T3 machine referred to such addresses as "symmetric" addresses, whereas other addresses, such as those allocated off of the stack as the application is running, are termed to be "non-symmetric". Symmetric addresses combined with the simple

```
static void initialize_shared_memory( void )
{
  extern VA_PML4T_ENTRY *KN_pml4_table_cpu[];
  int cpu;
  for( cpu=0 ; cpu < MAX_NUM_CPUS ;cpu++ ) {
    VA_PML4T_ENTRY * pml4 = KN_pml4_table_cpu[ cpu ];
    if( !pml4 )
      continue;
    KERNEL_PCB_TYPE * kpcb = (KERNEL_PCB_TYPE*) KN_cur_kpcb_ptr[cpu];
    if( !kpcb ) continue;
    VA_PML4T_ENTRY dirbase_ptr = (VA_PML4T_ENTRY)
      (KVTOP( (size_t) kpcb->kpcb_dirbase ) | PDE_P | PDE_W | PDE_U );
    int other;
    for( other=0 ; other<MAX_NUM_CPUS ; other++ ) {
      VA_PML4T_ENTRY * other_pml4 = KN_pml4_table_cpu[other];
      if( !other_pml4 ) continue;
      other_pml4[ cpu+1 ] = dirbase_ptr;
    }
  }
}
```

(a) Kernel code

```
static inline void * remote_address( unsigned core,
                                     volatile void * vaddr)
{
  uintptr_t addr = (uintptr_t) vaddr;
  addr |= ((uintptr_t) (core+1)) << 39;
  return (void*) addr;
}
```

(b) User code

1: Code to enable direct access shared memory (a), and to convert a local address to a remote address (b).

remote address translation function make it extremely easy for one process to read or write the corresponding data in another process' address space running on a different core of the same processor.

Replicating the top-level page table entry across all cores seems like a straightforward approach to allowing efficient data sharing between processes running on a multi-core processor as part of a parallel application. There has been much research in operating system support for shared memory. Our approach is similar to approaches that have been taken for single-address space operating systems. However, we have been unable to find any prior work where such direct access shared memory between all processes was desired or achieved.

4. POTENTIAL BENEFITS

Our direct access shared memory approach has several advantages over the existing approach for intra-node data movement in Catamount and also overcomes several limitations of using POSIX shared memory regions.

For MPI point-to-point communications, our approach allows for single-copy MPI transfers without the overhead of kernel involvement. Catamount's static page tables eliminate the need to remap pages on the fly. User-level access avoids any serialization through the kernel. In addition to the extra copies imposed by POSIX shared memory, an MPI implementation must deal with managing a fixed-sized shared region. This could become a more severe problem if core counts continue to increase faster than the amount of memory per node.

Our approach also supports functionality that none of the existing approaches for intra-node data movement do. MPI collective reduction operations can be performed directly in place at the destination buffer. Rather than having each process copy their operands into shared memory, perform the operation, and then copy the result out, direct access shared memory allows for having a process directly perform the operation on the corresponding process' buffer. One-sided operations, such as the MPI-2 remote memory access operations and Cray SHMEM get/put operations are also easily supported.

Direct access shared memory seems to be a natural fit for implementing a Partitioned Global Address Model, such as Unified Parallel C [5] or Co-Array Fortran [8]. These languages and libraries provide a partitioned address space that has "close" private memory and "far" shared memory. Because our approach can use loads and stores directly on a node, it may be possible to provide efficient compiler or library support within these environments.

5. SUMMARY

This paper has described a method of using page tables and virtual addressing to provide direct access shared memory between the processes running on a multi-core processor. This approach has many significant advantages over other methods of providing intra-node data movement. In particular, it can be used to implement MPI in a way that avoids any extraneous memory-to-memory copies – which will become extremely important as core counts increase. It can also be used to implement functionality that POSIX-style shared memory cannot – such as one-sided data get/put operations and in-place reduction operations.

Our implementation of direct access shared memory also emphasizes the simplicity of a lightweight kernel environment. The kernel code to enable direct access shared memory is approximately 32 lines. A similar strategy could be implemented in a more full-featured operating system like Linux, but would likely involve significant changes to the internal memory management code and structures.

Direct access shared memory support has been integrated into CNW version 2.0.41 and has been tested on more than three thousand nodes of the Red Storm machine. We are currently making the necessary changes to MPI to make use of this new capability and expect to have performance results shortly.

6. ACKNOWLEDGMENTS

The implementation of direct access shared memory in Catamount would not have been possible without Trammell Hudson and John Van Dyke. Kevin Pedretti and Kurt Ferreira also provided many useful discussions regarding virtual memory and lightweight kernel memory management. The author also gratefully acknowledges the support of Sue Kelly and the Cray support staff at Sandia.

7. REFERENCES

[1] R. Brightwell, W. Lawry, A. B. Maccabe, and R. Riesen. Portals 3.0: Protocol building blocks for low overhead communication. In *Proceedings of the 2002 Workshop on Communication Architecture for Clusters*, April 2002.

[2] Ron Brightwell, Trammell Hudson, Kevin T. Pedretti, and Keith D. Underwood. SeaStar interconnect:

Balanced bandwidth for scalable performance. *IEEE Micro*, 26(3), May/June 2006.

[3] Darius Buntinas, Guillaume Mercier, and William Gropp. Data transfers between processes in an smp system: Performance study and application to mpi. In *Proceedings of the 2006 International Conference on Parallel Processing*, August 2006.

[4] William J. Camp and James L. Tomkins. Thor's hammer: The first version of the Red Storm MPP architecture. In *In Proceedings of the SC 2002 Conference on High Performance Networking and Computing*, Baltimore, MD, November 2002.

[5] Willaim W. Carlson, Jesse M. Draper, David E. Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, May 1999.

[6] Suzanne M. Kelly and Ron Brightwell. Software architecture of the light weight kernel, Catamount. In *Proceedings of the 2005 Cray User Group Annual Technical Conference*, May 2005.

[7] Arthur B. Maccabe, Rolf Riesen, and David W. van Dresser. Dynamic processor modes in Puma. *Bulletin of the Technical Committee on Operating Systems and Application Environments (TCOS)*, 8(2):4–12, 1996.

[8] Robert W. Numrich and John Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[9] Lance Shuler, Chu Jong, Rolf Riesen, David van Dresser, Arthur B. Maccabe, Lee Ann Fisk, and T. Mack Stallcup. The Puma operating system for massively parallel computers. In *Proceeding of the 1995 Intel Supercomputer User's Group Conference*. Intel Supercomputer User's Group, 1995.

[10] Stephen R. Wheat Timothy G. Mattson, David Scott. A TeraFLOPS Supercomputer in 1996: The ASCI TFLOP System. In *Proceedings of the 1996 International Parallel Processing Symposium*, 1996.